Getting the Most out of the Intel® Itanium® Architecture

Compiler Optimization and Performance Tuning

Dr. Gernot Hoyler
Technical Marketing
Intel® EMEA
mailto:Gernot.Hoyler@intel.com

intel

Agenda

- Optimizing with the Intel® Compilers
- Using Intel Performance Libraries
- Performance Tuning with Intel VTune™
- Other Optimization Opportunities
- Summary

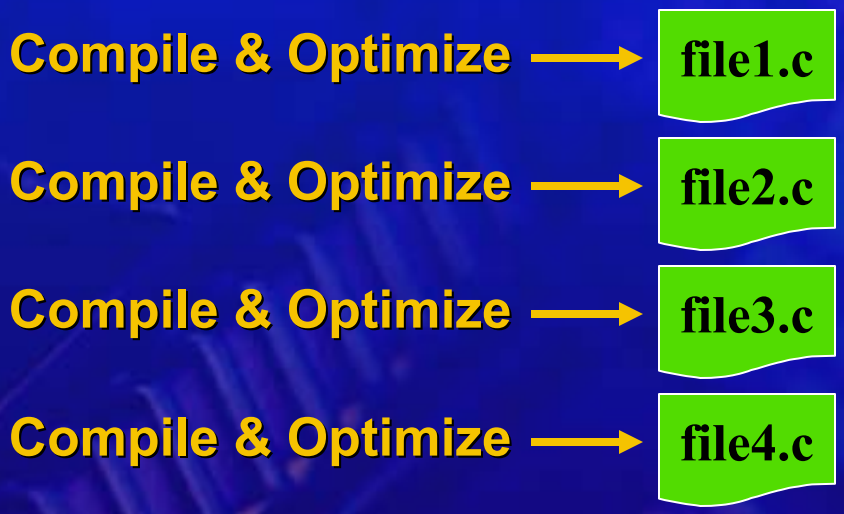
Compiler Optimization Flags

- -O0: disables optimization
- -O1: optimizes for speed without increasing code size
- -O2: optimizes for speed (default)
- -O3: enables -O2 plus more aggressive optimizations, may not improve performance for all programs
- -tpp2: Itanium® 2 Code Generation (instruction mix)
- -fno-alias: assumes no aliasing in program (may be unsafe)
- -align: analyzes and reorders memory layout for variables and arrays (FTN only)
- -pad: enables changing variable and array memory layout (FTN only)

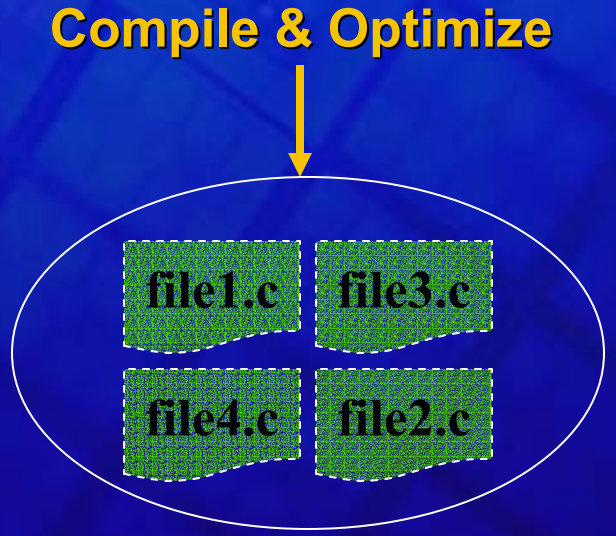
Interprocedural Optimization

Extends optimizations across file boundaries.

Without IPO (or with -ip)



With IPO



IPO Benefits

- Exposes more optimization opportunities (constant propagation, function/data promotion, data layout optimizations, etc.)
- Optimizes function ordering
- Reduces function call overhead by inlining functions across file boundaries
- Can significantly reduce code size through dead code elimination
- Available for both Itanium® and IA-32

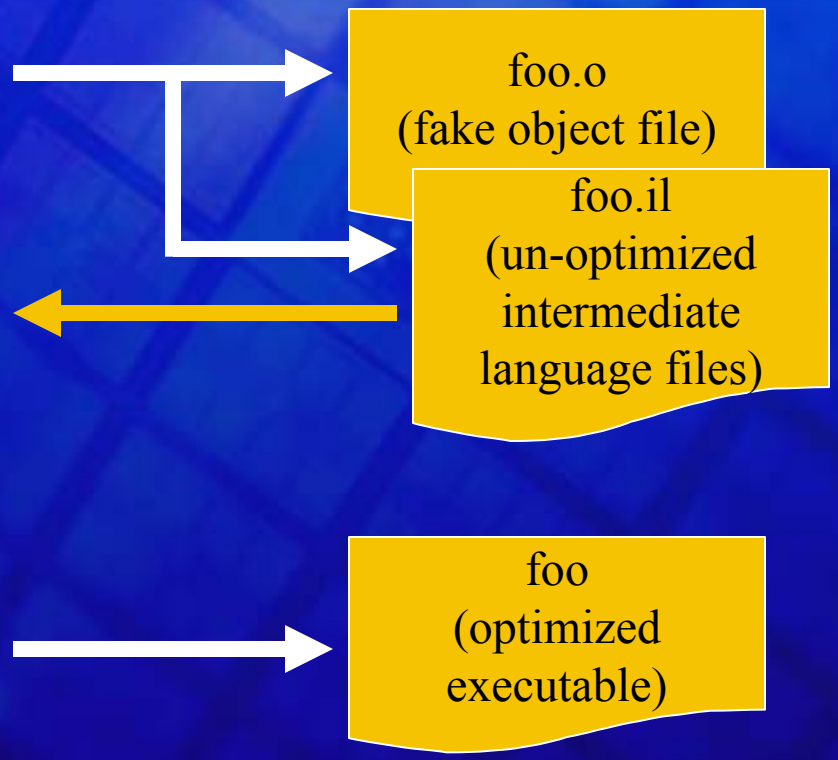
How IPO Works

1. Compile program
   `icc -c -ipo foo.c`

2. Link program
   `icc -o foo -ipo foo.o`

   2a. Compiler performs whole-program optimizations

   2b. Compiler invokes linker to produce executable



Programs that Benefit from IPO

- Many small utility functions
- Frequent constructor/destructor invocation
- One-liner member functions
- Lynx success story
  - Intel® Spice-like circuit simulator
  - Highly tuned algorithmically
  - Intel compiler (icc) with O2 & IPO:
    - 1.2x - 5.2x speedup over gcc -O (2x typical)
    - 1x - 2.4x speedup over icc -O2 (1.2x typical)

Profile-Guided Optimization

Feed back of profile data gathered during program execution to improve subsequent builds.

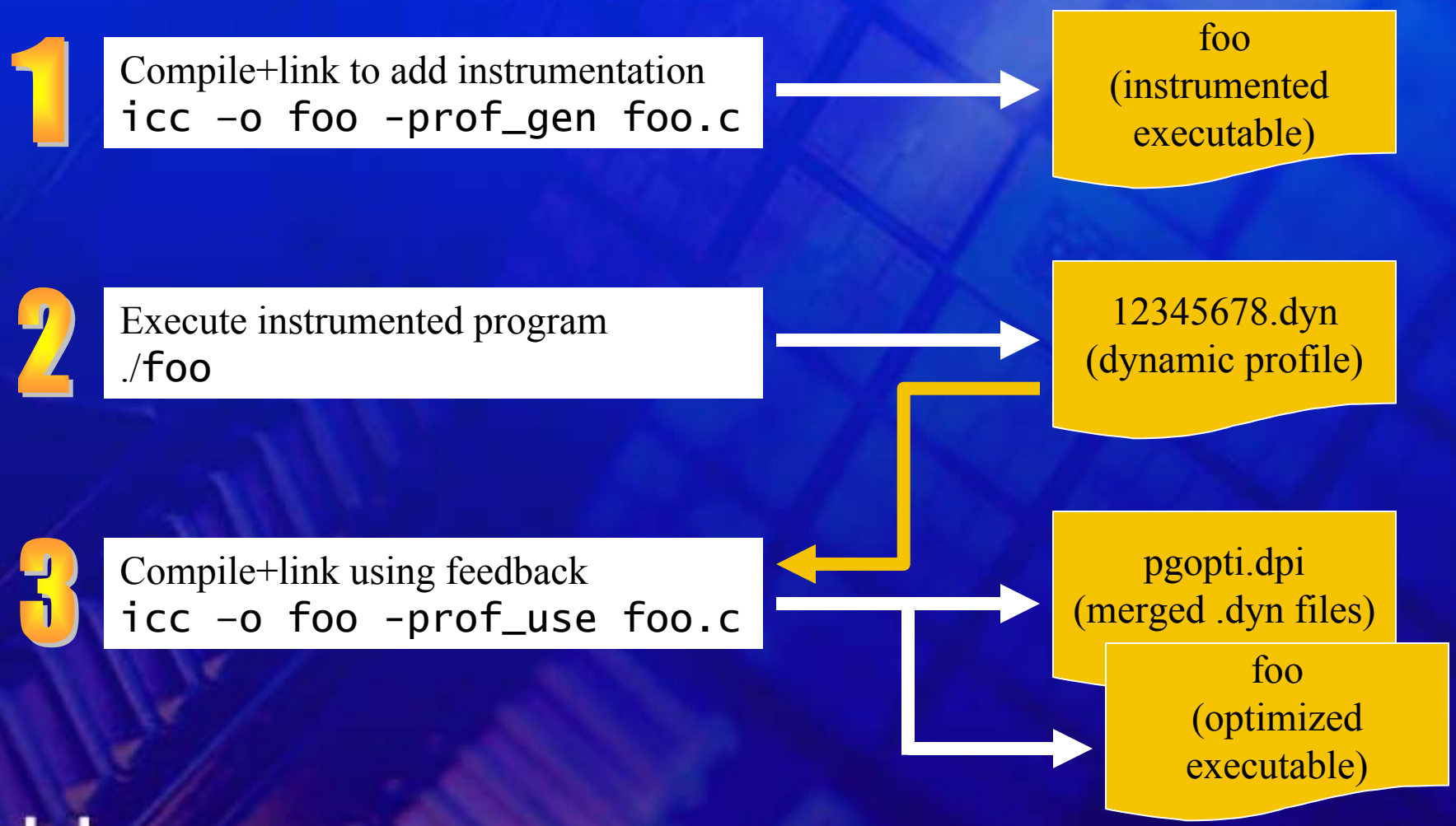
Benefits:

- More accurate branch prediction
- Better register allocation
- Improved IPO inlining
- Basic block movement

```
status = UtilityFunc (arg1, arg2, arg3);
if (status != 0) // Not expected to fail
   HandleErr (status);
```

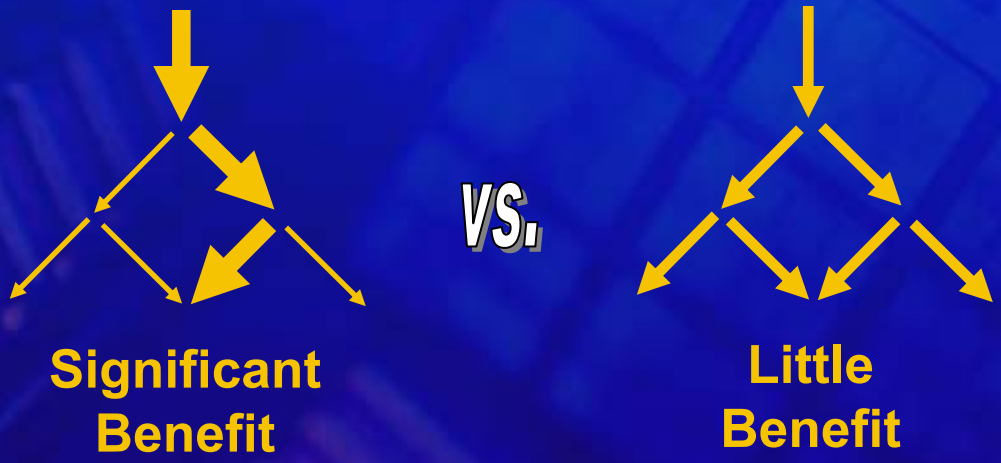
- Improves I-cache behavior
- Available for Itanium® and IA-32

How PGO Works



Programs that Benefit from PGO

- Consistent hot paths
- Many if statements or switches
- Nested if statements or switches



Practical Considerations

- It is not necessary to regenerate profile with every build
- Performance benefits degrade gradually as source code changes
- Regenerate profile when justified
- Source files must stay in same dir from -prof_gen to -prof_use (full source paths recorded in profile files)

PGO makes IPO work better

Compiler Directives/Pragmas

- !DIR$ IVDEP : no loop-carried dependency
- !DIR$ SWP : software-pipeline the loop
- !DIR$ NOSWP
- !DIR$ UNROLL (*n*) : unroll a counted loop
- !DIR$ NOUNROLL
- !DIR$ LOOP COUNT (*n*) : loop count is likely to
- !DIR$ DISTRIBUTE POINT : perform loop distribut.

Use in conjunction with optimization report flags (-opt_report)

Compiler Directives/Pragmas

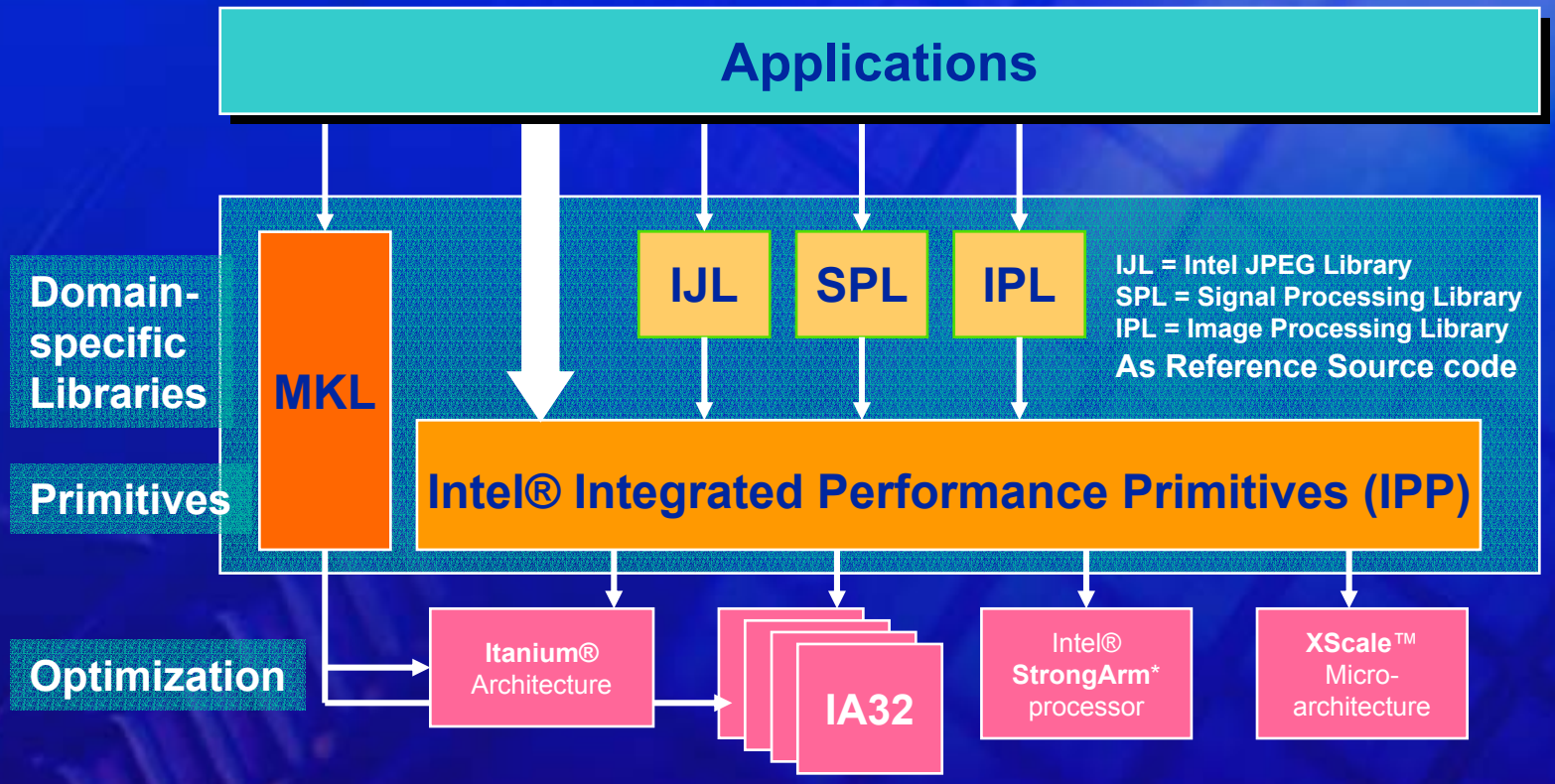
- !DIR$ PREFETCH *a* : insert prefetch instructs.
- !DIR$ NOPREFETCH *a*
- !DIR$ PARALLEL : disambiguates assumed data dependencies
- !DIR$ NOPARALLEL

Use in conjunction with optimization report flags (-opt_report , -par_report)

Agenda

- Optimizing with the Intel® Compilers
- Using Intel Performance Libraries
- Performance Tuning with Intel VTune™
- Other Optimization Opportunities
- Summary

Intel® Performance Libraries



Expanded functionality & full platform integration

Intel® Performance Libraries

- Performance: raison d'etre
- Resisted adding software only for functionality
- Finds ways to exploit every part of system: processor, memory, multiple processors, multiple nodes
- Makes high performance easy on lots of scientific, engineering and math codes
- Example: DGEMM (part of MKL) achieves
  - 95 % of theoretical peak performance on Itanium® 2
  - 75-77 % of theoretical peak performance on Xeon™

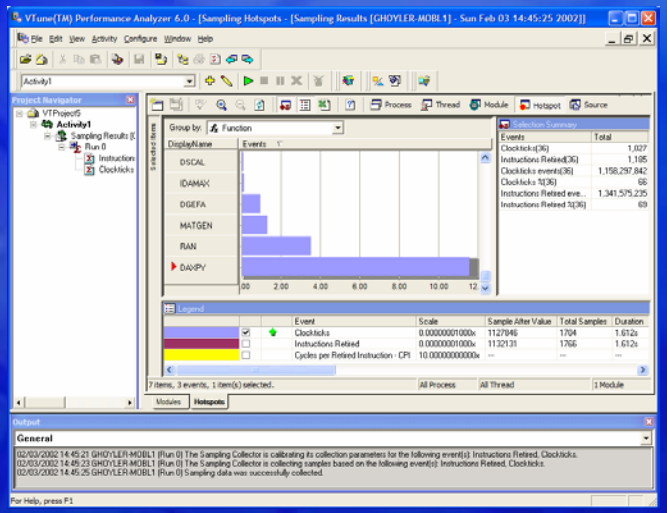
intel

Agenda

- Optimizing with the Intel® Compilers
- Using Intel Performance Libraries
- Performance Tuning with Intel VTune™
- Other Optimization Opportunities
- Summary

VTune™ Overview

- Graphical performance tool
- Various experiments
  - Counter Monitoring (OS counters)
  - Call Graph (who calls who)
  - Sampling (standard hot spot profiling & HW counters)



- Visualization of results as graphs and/or tables
- GUI running under Windows* (32 or 64 bit)
- Remote collectors for Windows & Linux*
- Commercial product, list price: 699 US$
- Current Version 6.1 adds Itanium® 2 support
  http://developer.intel.com/software/products/vtune/vtune61/index.htm

intel

VTune™ Architecture

Local Analysis

Both data collector and GUI on the same system

Remote Analysis

Data collector

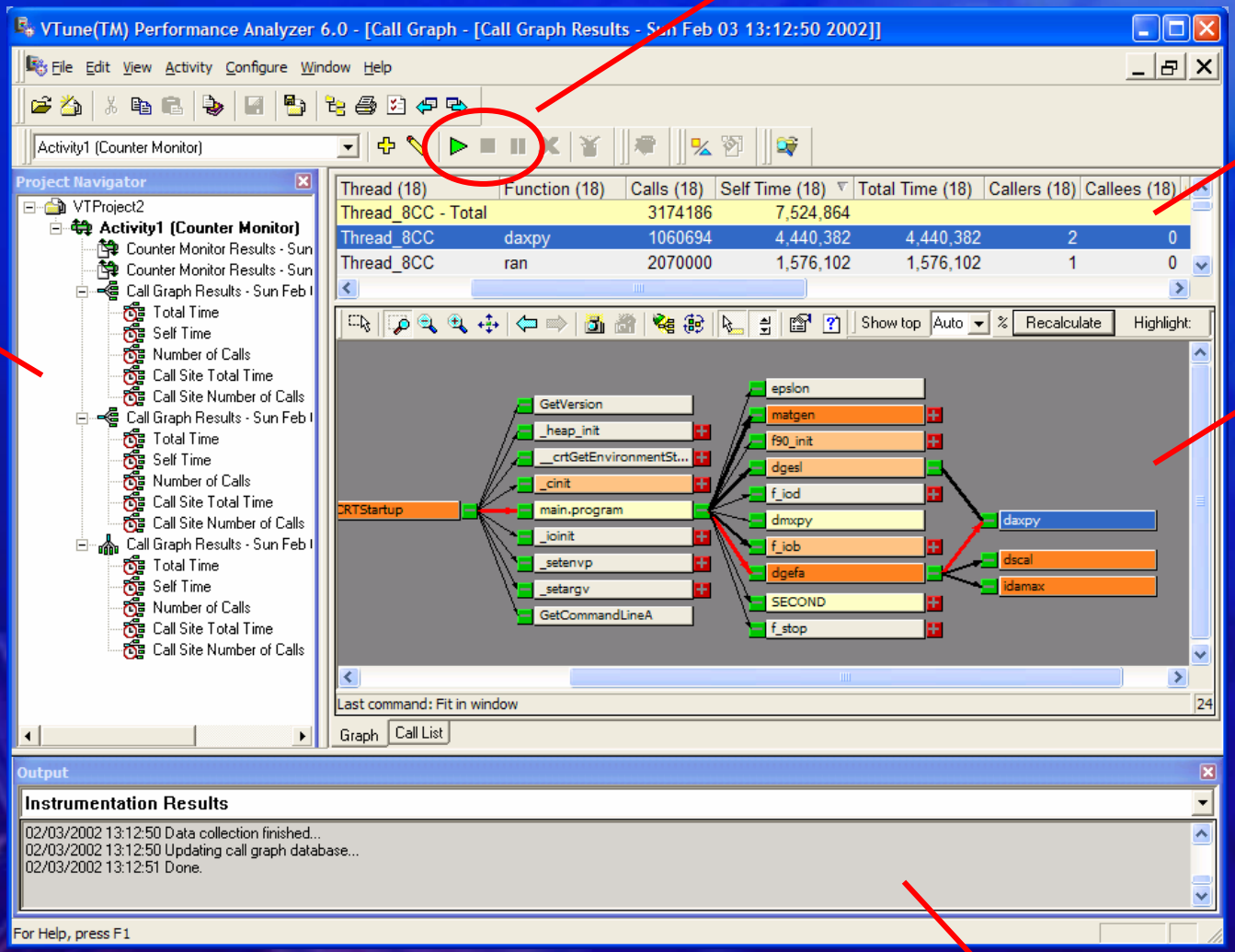
Data collector

GUI

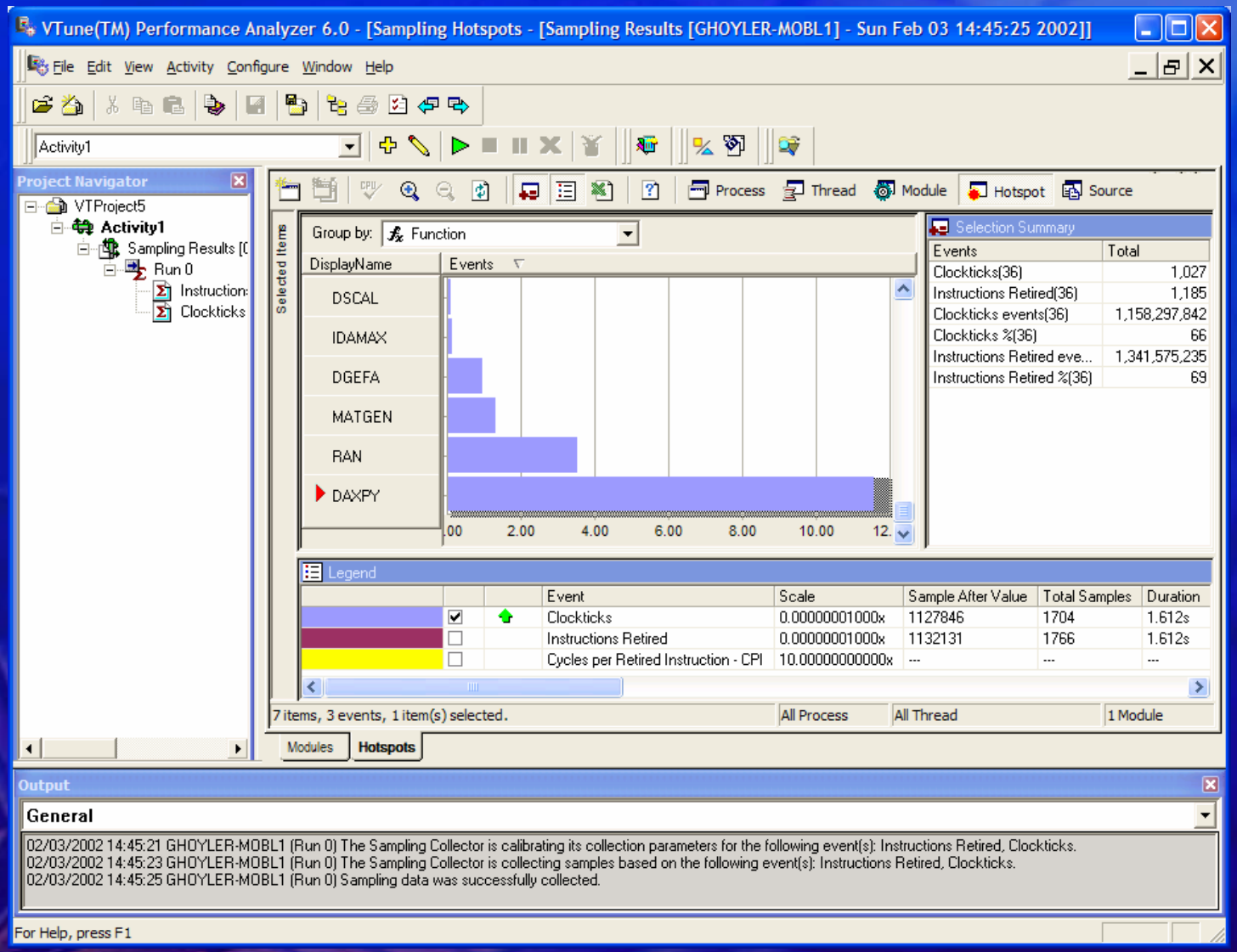
Linux* Data Collectors

- VTune™ includes binary kernel modules which are precompiled for various 2.4 and 2.5 Linux kernels
- Need to recompile kernel for Linux-64 (minor source code changes necessary)
- Install VTune™ 6.1 on a Windows client
- Install VTune Remote Collector package on the remote Linux system
- Copy or share the source code with client
- Open a new project on the Windows client and connect to the remote Linux system

The VTune™ GUI

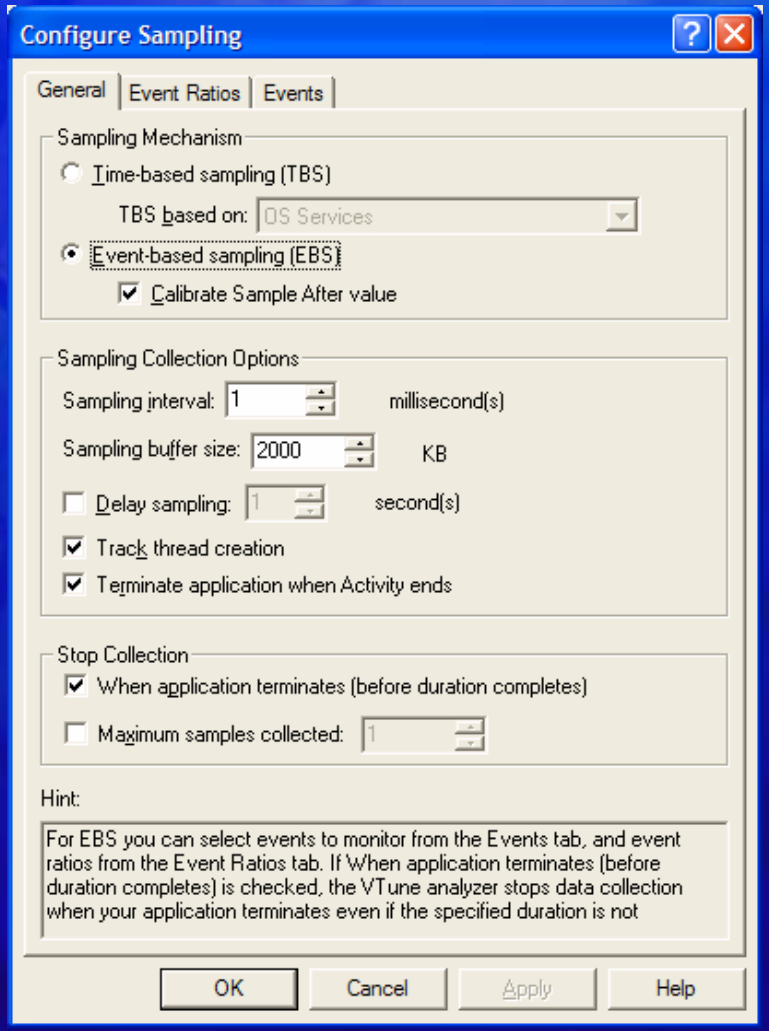


VTune™ Sampling

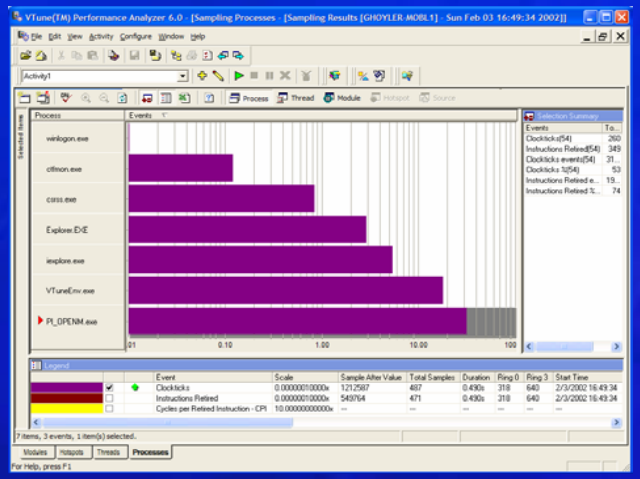


Sampling Methods

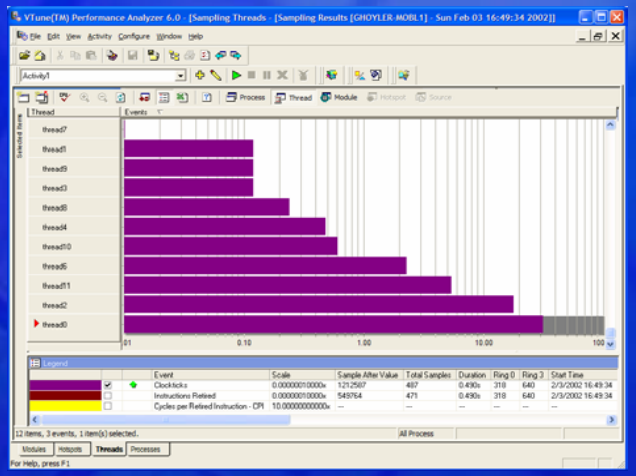
- Time-based sampling (TBS)
  - Uses OS timer
  - Delivers hotspot profile
  - Available on almost every CPU, including mobile Intel® & AMD* processors
- Event-based sampling (EBS)
  - Uses processor counters
  - Much more powerful than simple time-based sampling
  - Delivers hotspot profile + CPU specific data (FP instructions, cache misses, bus activity, etc.)



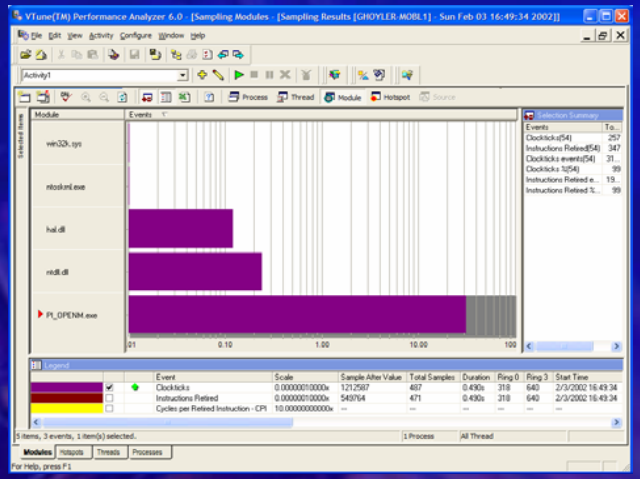
Multilevel Views ...



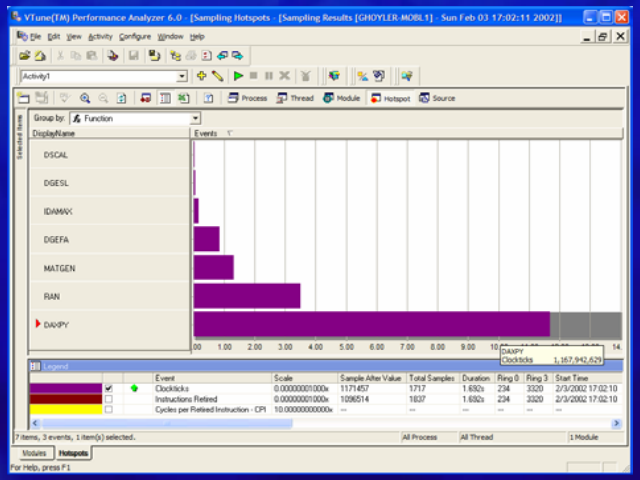
Process level



Thread level



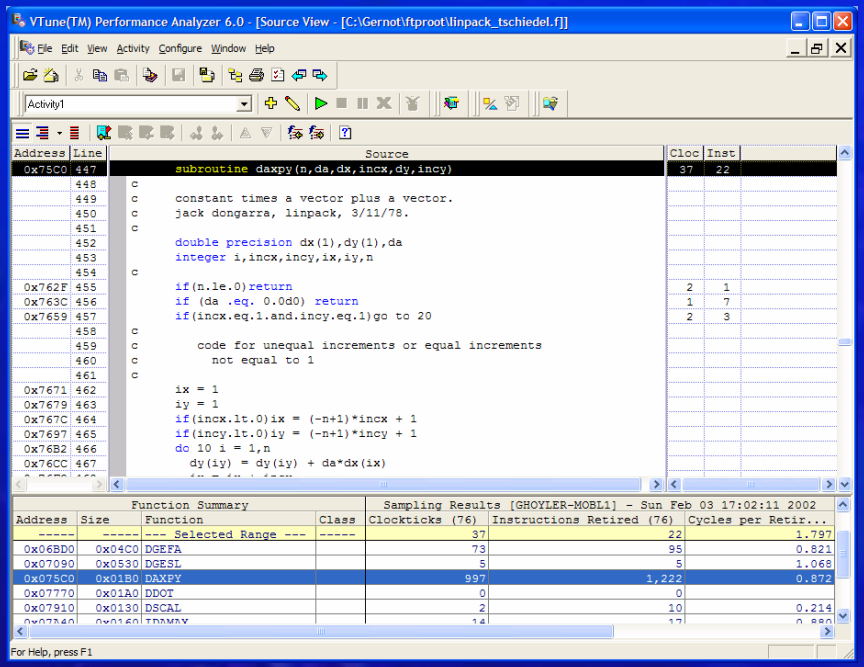
Module level

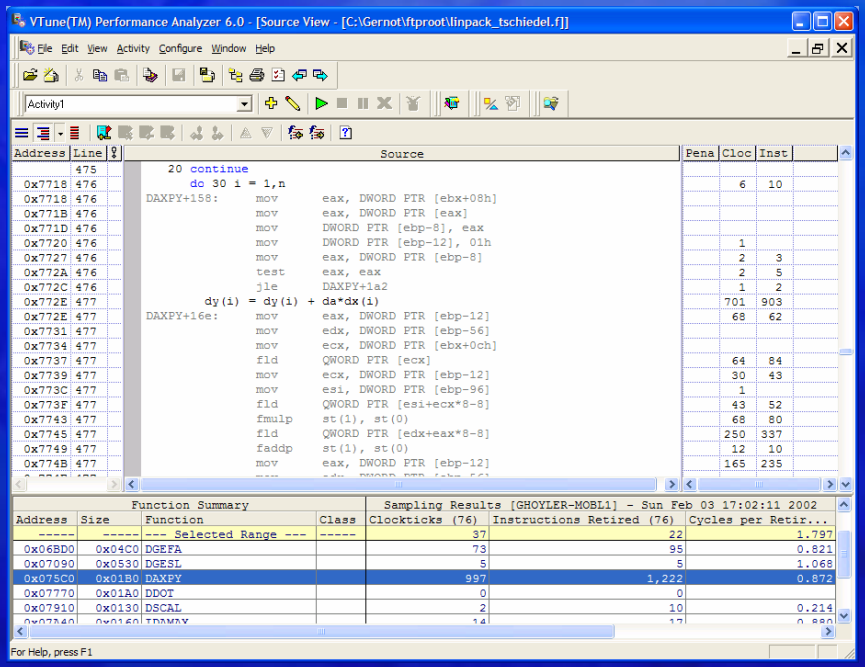


Function level

...

… down to source & assembly!

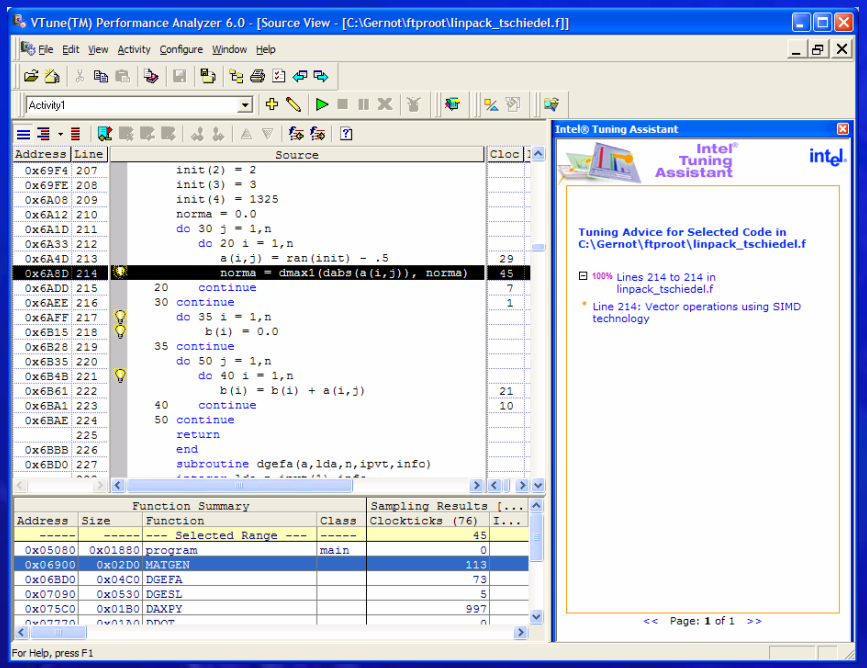


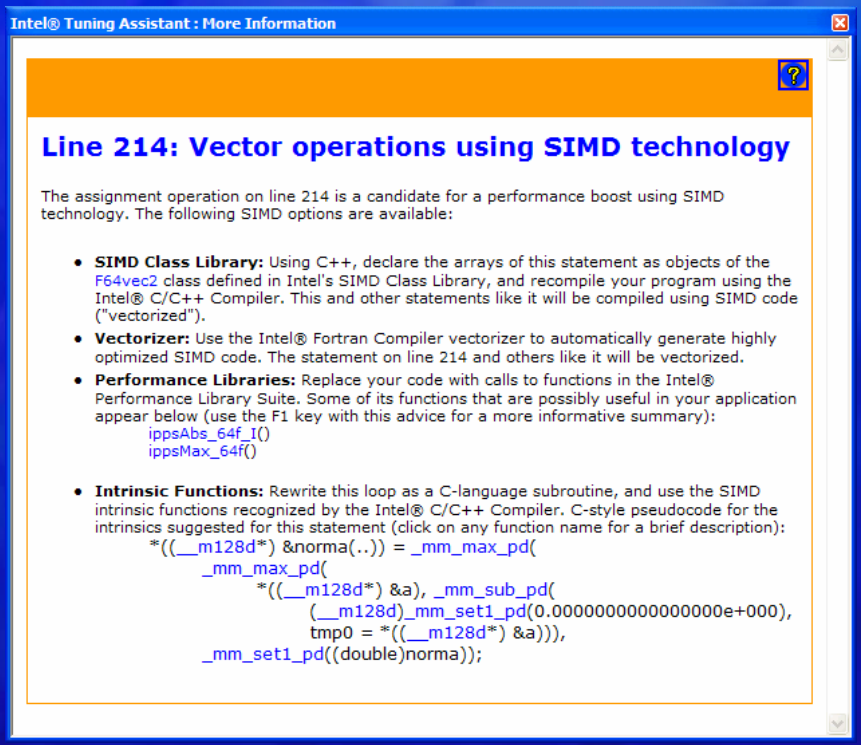


Source code level → ***Assembly level***

Source code level analysis requires use of debugging flag /Zi,-g in order to include symbolic source line info in the binary!

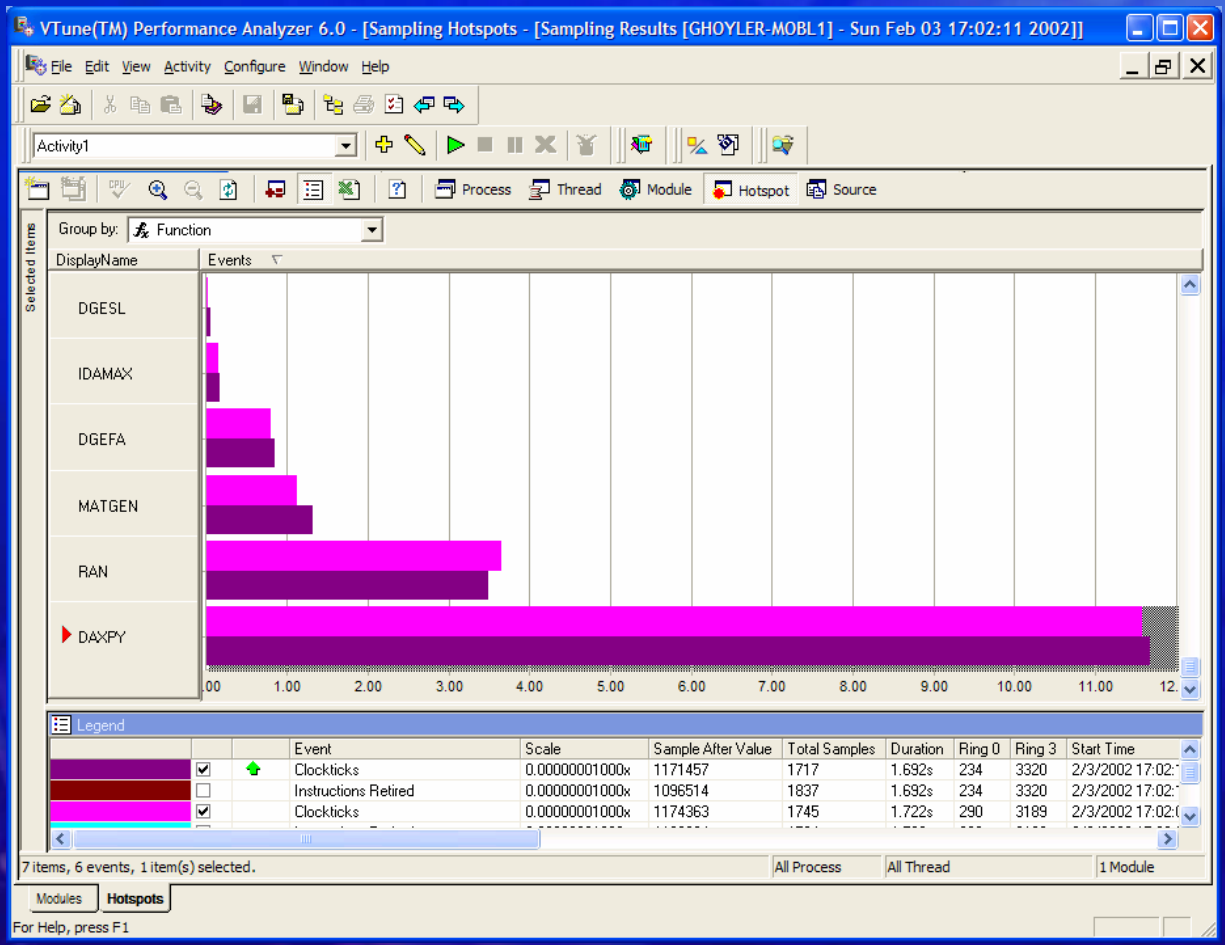
Intel® Tuning Assistant





Processor specific expert system – based on source code analysis!

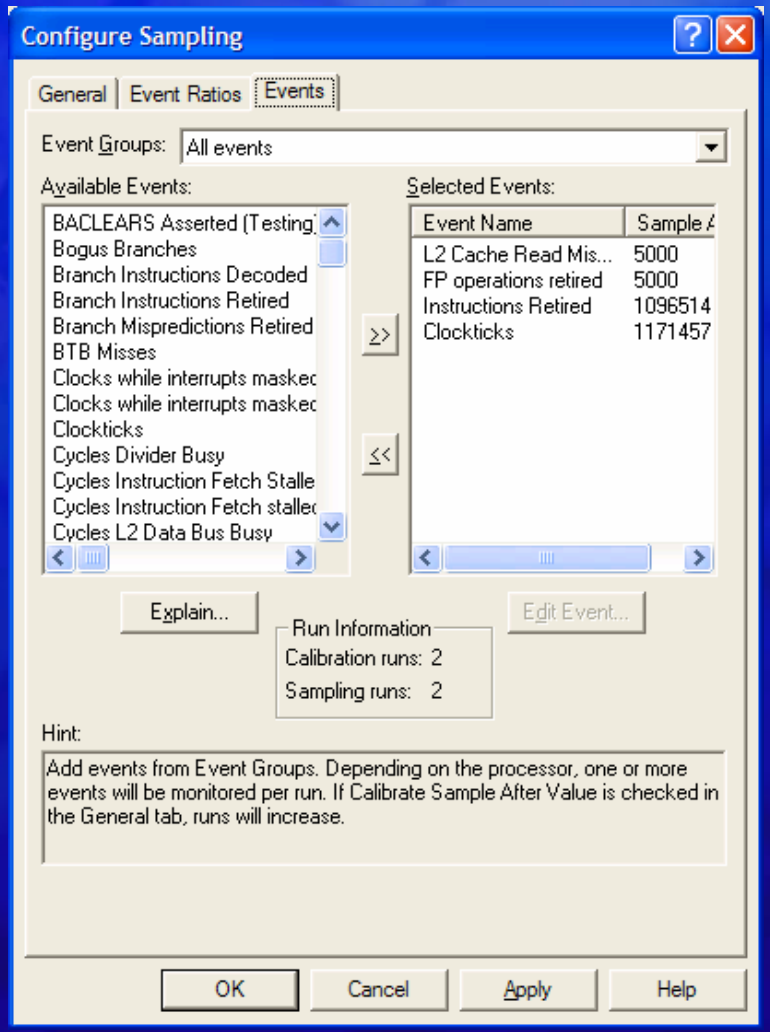
Multiple Sampling Activities



Allows direct comparison of multiple profiles

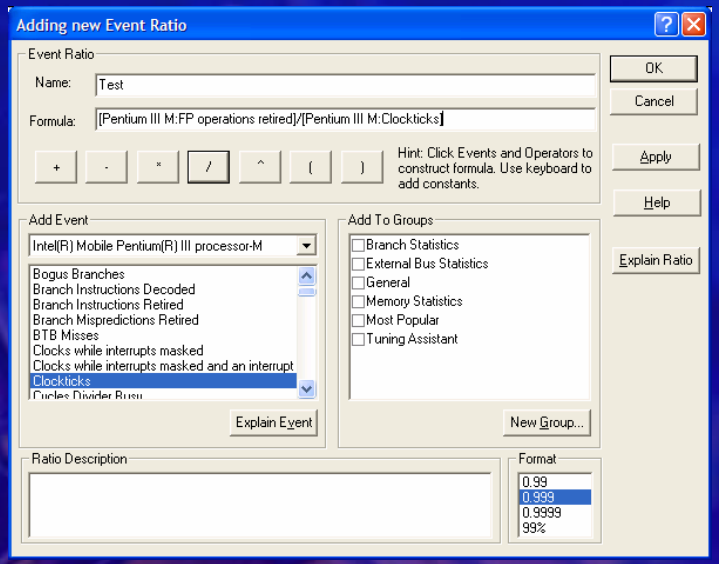
CPU Specific HW Counters

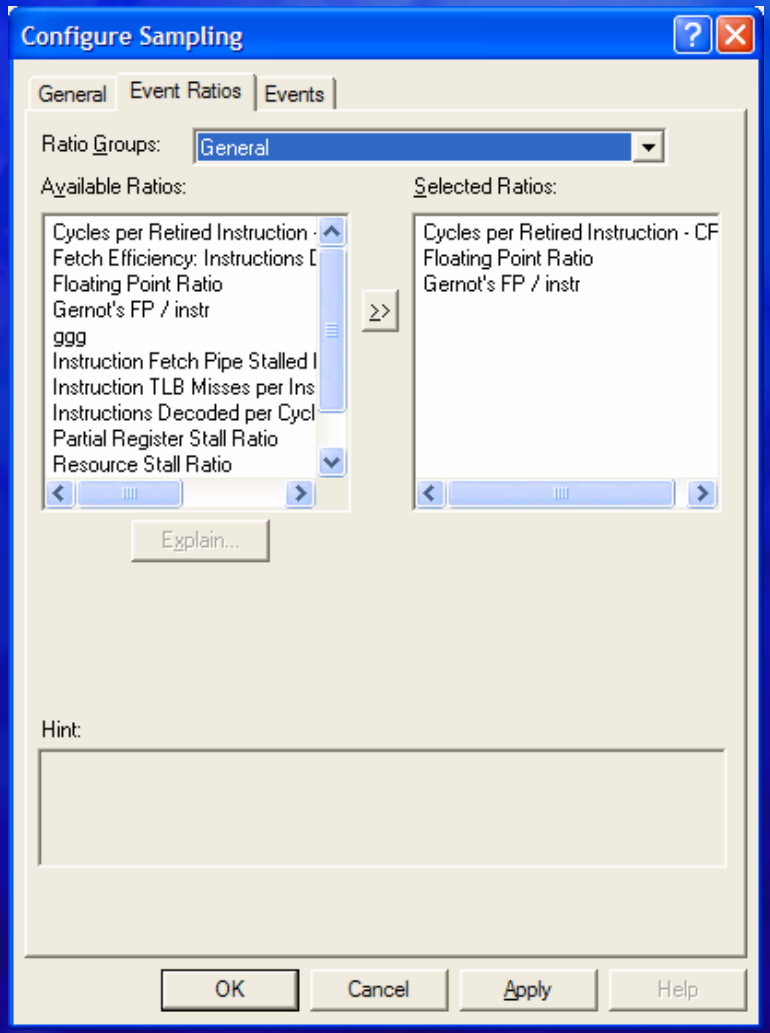
- Event based sampling allows programming of CPU specific counters
- Useful to look at cache, memory and TLB activities, FP throughput as well as other CPU characteristics
- Can require multiple runs to collect all data!



Counter/Event Ratios

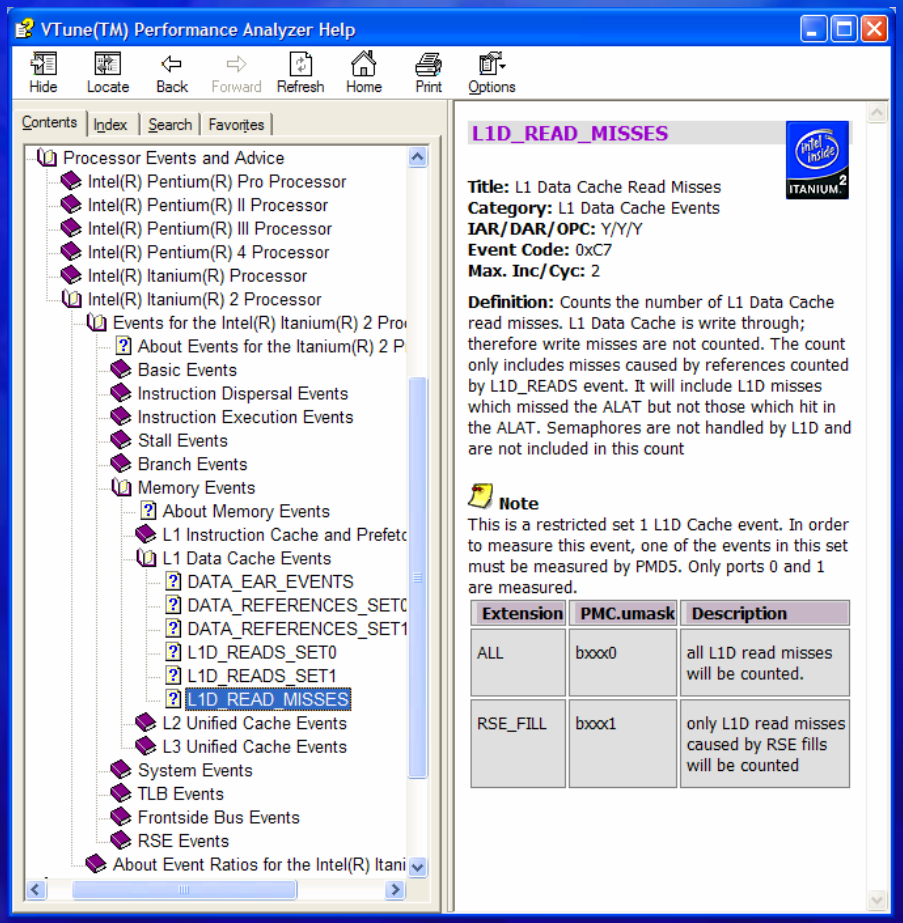
- Predefined standard ratios available
  - e.g. CPI (clocks per instr.)
- User defined ratios can be easily added
  - even complex formulae possible





Which Events and Event Ratios should be Monitored?

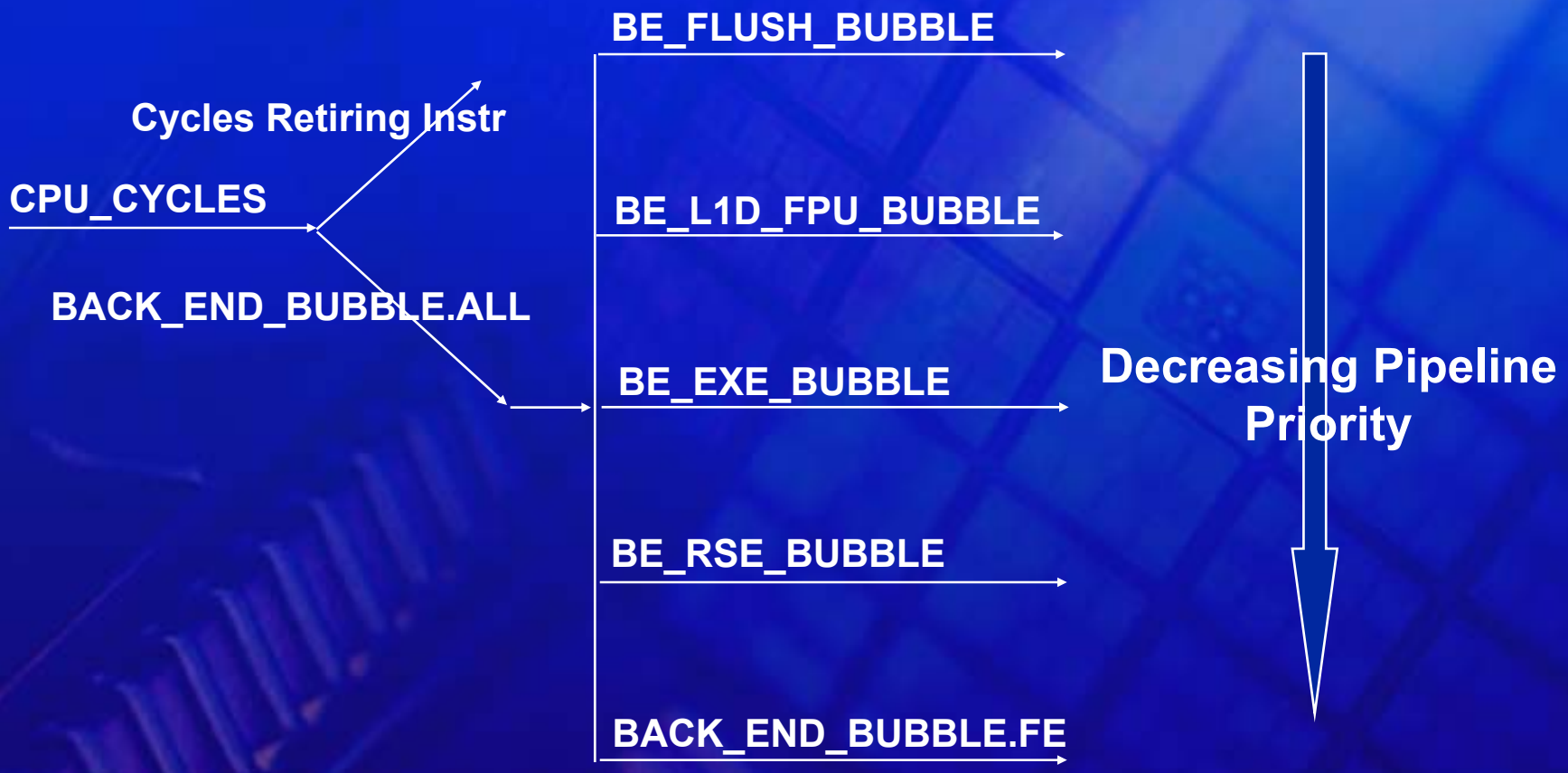
- Consult the VTune™ online manual for a list of primary events and event ratios to start with, afterwards extend and refine as needed
- Itanium®: start with Basic, Instruction, Execution, TLB, Memory and Branch Events and use Stall Events for cycle accounting
- NetBurst™: start with „Primary Performance Tuning Events“ and ratios, note that an impact of > 2 might already be critical



Cycle Accounting on Itanium®

- In general pipeline is stalling and not issuing instructions to functional units on every cycle; Cycles are wasted waiting for resources like e.g.
  - data from memory or cache
  - instructions to be processed
  - register availability due to excessive RSE activity
- Use sub-set of PMU counters to find main stall source
- Hierarchical tree structure enables systematic performance methodology

Tree Structure of Stall Cycles

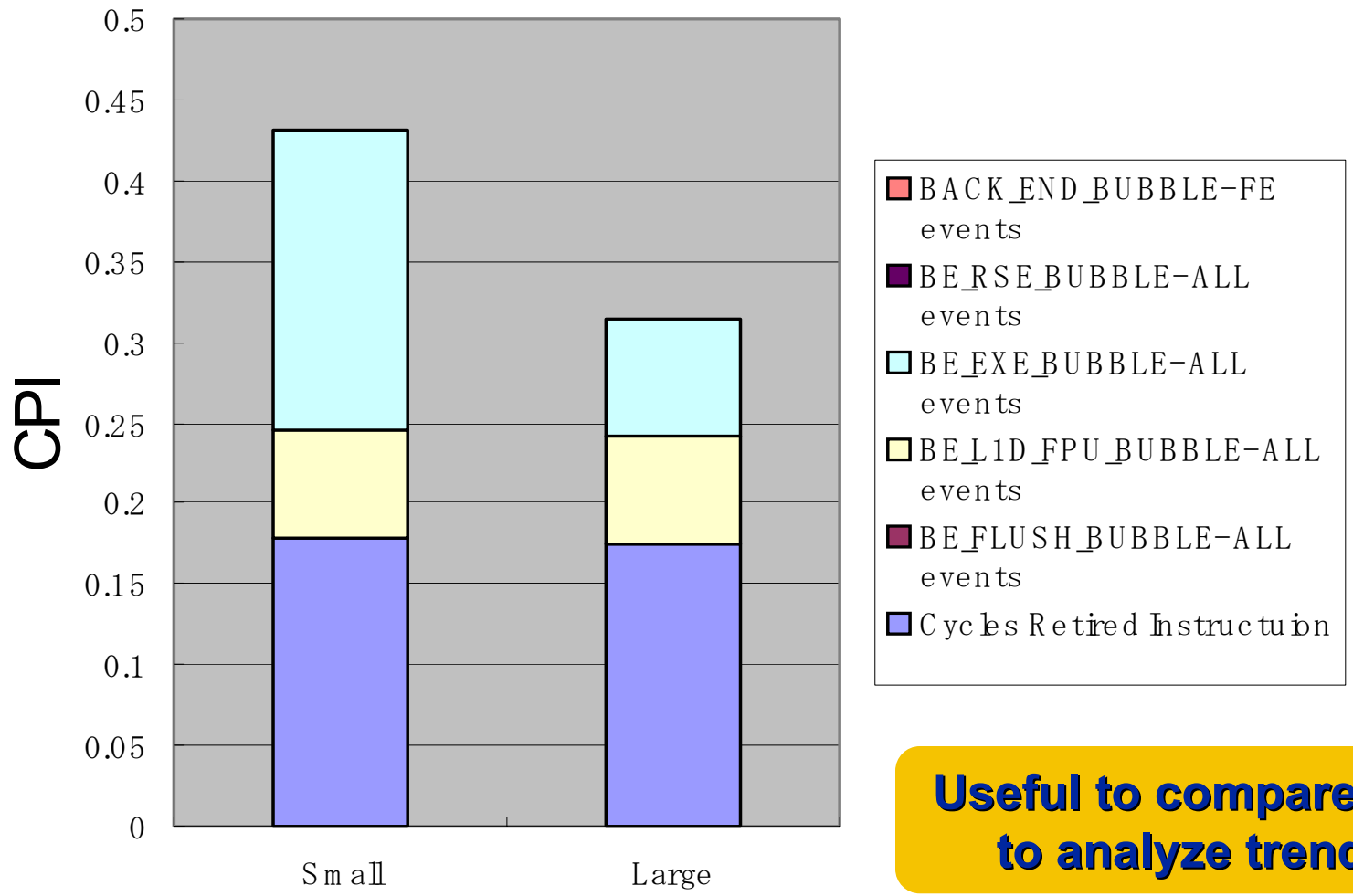


Cycle accounting sum rule starts the decomposition

E.g.: Components of BE_L1D_FPU_BUBBLE

Sub-events of BE_L1D_FPU_BUBBLE have multiple levels

Cycle Accounting Chart



Useful to compare and to analyze trends

Agenda

- Optimizing with the Intel® Compilers
- Using Intel Performance Libraries
- Performance Tuning with Intel VTune™
- Other Optimization Opportunities
- Summary

Other Optimization Opportunities

- Use of command line tools such as `pfmon`, `hpcmon`, `vtl` etc.

Konsole - gernot@localhost:~ - Konsole

```
================================================================================
HPCmon Ver. 1.1 - (c) 2002 Gernot Hoyler, Intel GmbH
[pfmon Ver. 1.1 - (c) 2001-2002 Hewlett-Packard Company]
================================================================================
CPU speed              :       897512179 Hz
Elapsed time           :      8036620308 cycles (8.95 sec)
Monitored clockticks   :      8035906015 cycles (8.95 CPU sec)
Instructions retired   :      6657720795 events (743.52 Mev/sec, 0.83 IPC)
Nops retired           :       415678786 events (46.42 Mev/sec, 0.05 NPC)
Off predicated instr.  :          304378 events (0.03 Mev/sec, 0.00 NPC)
FP instr. retired      :      1608094008 events (179.59 Mev/sec or MFLOPS)
FP results FTZ'ed      :               0 events (0.00 Mev/sec)
2nd lev. cache refs.   :      3071626791 events (343.03 Mev/sec)
2nd lev. cache misses  :       252098578 events (28.15 Mev/sec, 91 % hit rate)
3nd lev. cache refs.   :       353178614 events (39.44 Mev/sec)
3nd lev. cache misses  :       252082821 events (28.15 Mev/sec, 28 % hit rate)
L2 DTLB misses         :          495705 events (0.06 Mev/sec)
L2 ITLB misses         :              62 events (0.00 Mev/sec)
Bus data cycles        :      1412740182 events (157.77 Mev/sec 78 % bus load)
Bus snoop stall cycles:       223012470 events (24.91 Mev/sec)
Misaligned loads       :               0 events (0.00 Mev/sec)
Stores to shared line  :            9366 events (0.00 Mev/sec)
Branches retired       :       302463901 events (33.78 Mev/sec)
Mispredicted branches  :           82394 events (0.00 Mev/sec, 99 % hit rate)
%
```

Can help to identify obvious bottlenecks at a first glance

Other Optimization Opportunities

- Use of hand coded assembly
  - Quite tricky to apply to larger code portions
  - Consult IA-64 guides for instruction mix and latencies
  - GCC inline assembly syntax not supported, use asm intrinsics instead
- Give different OS & Compilers a chance
- Submit the code to specialists at your OEM or at Intel® (e.g. Solution Centers)

Further information & tricks can be found in the Itanium® 2 Processor Reference Manual for Software and Optimization

http://developer.intel.com/design/itanium/manuals.htm

Summary

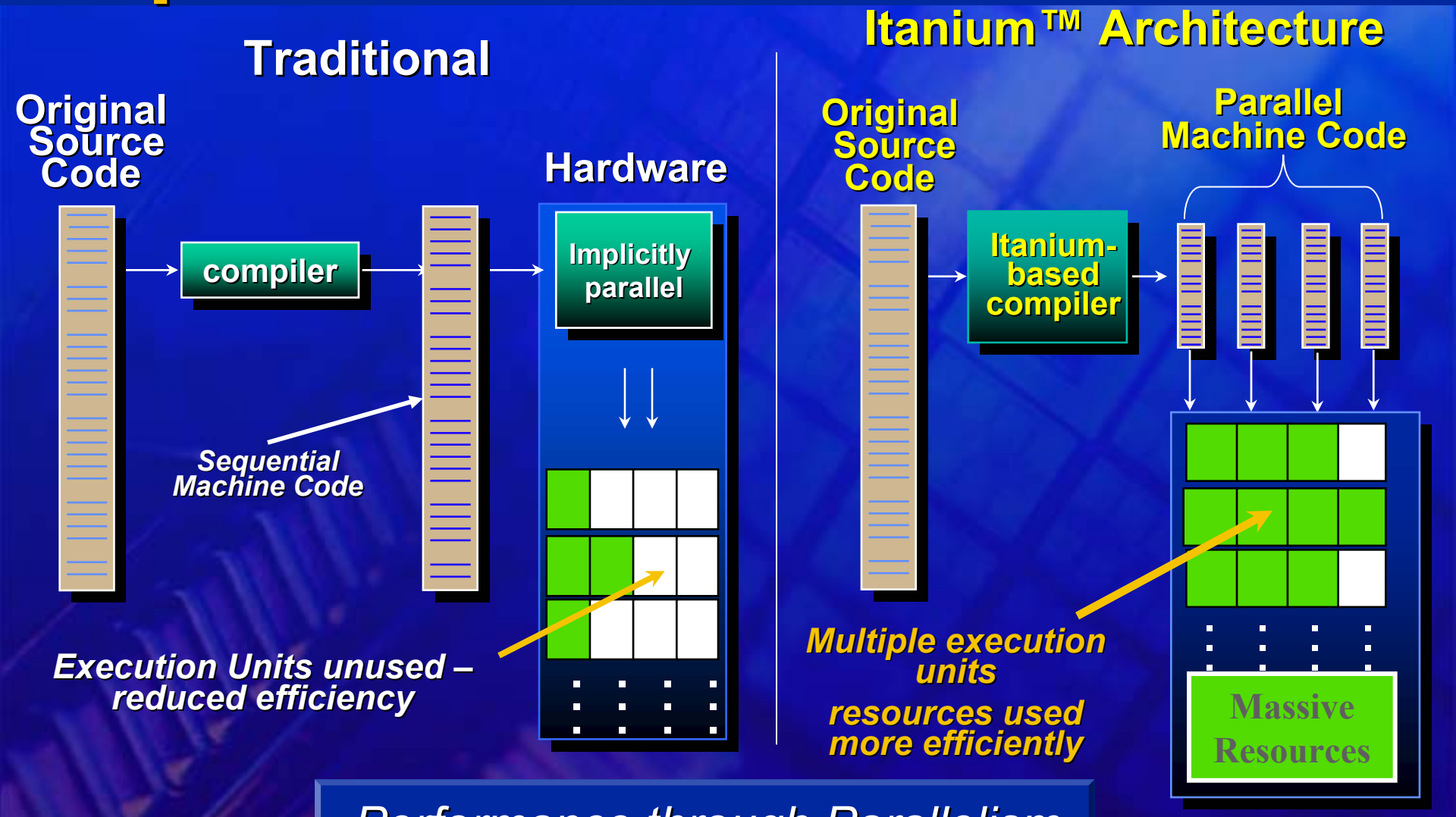
- The Intel® Itanium® Architecture is still quite a new µarch with a huge performance potential
- Always use the latest versions of Intel® tools
- Apply standard compiler optimization flags plus IPO & PGO
- Use performance libraries wherever possible
- For advanced optimizations take the Intel® VTune™ Performance Analyzer and identify the bottlenecks
- Use hand coded assembly only as a last resort

Thank You !

www.intel.com/go/hpc

intel

Itanium™ Architecture
Explicit Parallelism



Performance through Parallelism

Block Diagram (Itanium 2)